NASA-CR-203342

VIS-AD DATA MANAGEMENT

William Hibbard[1&2], Charles R. Dyer[2] and Brian Paul[1]

[1]Space Science and Engineering Center
[2]Department of Computer Sciences
University of Wisconsin - Madison

1. INTRODUCTION

The VIS-AD system was designed as an algorithm development system, enabling scientists to visualize the results of experiments with their data analysis algorithms. However, VIS-AD could also become a very powerful data management system. VIS-AD provides a programming language similar to C for expressing scientific algorithms. The language provides a mechanism for users to define complex data types for the data objects of their algorithms. Data types can be defined for images, multi-spectral images, image sequences, gridded data, randomly located data, geometrical data such as boundary lines, and virtually any other data structures used in earth science. Furthermore, VIS-AD manages all these data types in a uniform way, and provides access to data through its programming language. These features could form the basis for a new way to manage environmental data.

2. DATA TYPES

VIS-AD enables its users to define data types for the data objects of algorithms, as follows. Define T as the set of types for the data objects in an algorithm. It is common for a programming language to define a set of primitive types (e.g. *int*, *real*), and to define a set of type constructors for building the types in T from the primitive types. We modify this by interposing a finite set S of scalar types between T and the primitive types. We define the primitive types as:

$$PRIM = \{int, string, real, real2d, real3d\}$$

where *real2d* and *real3d* are pairs and triples of real numbers. The user defines a finite set S of scalar types, and binds them to the primitive types by a function $P{:}S \rightarrow PRIM$. An infinite set T of types can be defined from S by:

$$S \subset T$$
$$(\text{for } i = 1,\ldots,n. t_i \in T) \Rightarrow (t_1,\ldots,t_n) \in T$$
$$(s \in S \wedge t \in T) \Rightarrow (s \rightarrow t) \in T$$

where $(t_1,\ldots,t_n)$ is a tuple type constructor with element types t_i, and $(s \rightarrow t)$ is an array type constructor with value type t and index type s.

The important consequence of the use of scalar types is that every primitive value, including an array index value, occurring in a data object of type $t \in T$, has a scalar type in S. The names of primitive values are a form of ancillary information, and the scalar types are a way of requiring the user to supply this ancillary information with data objects. The VIS-AD system uses this ancillary information to intelligently generate graphical depictions of data objects, but it may also be useful for supporting other intelligent data management functions.

3. EXAMPLES OF DATA TYPES

The VIS-AD system provides a simple syntax for data type definitions. We offer examples from an algorithm for discriminating clouds in GOES images. The following are examples of how the user defines scalar types in S and the function $P{:}S \rightarrow PRIM$:

```
type brightness = real;
type temperature = real;
type earth_location = real2d;
type image_region = int;
type time = real;
type count = int;
```

Here *brightness* and *temperature* are the visible and infrared radiance values of pixels in satellite images, *earth_location* is a pair of values for the latitude and longitude of pixel locations, *image_region* is an index into rectangular sub-images, *time* is an index for image sequences, and *count* is used for histograms.

The following are examples of how the user defines complex types in T (the keyword *structure* is used to indicate the tuple constructor):

```
type visir_image =
  array [earth_location] of
    structure {
      .visir_ir = temperature;
      .visir_vis = brightness;
    };

type visir_set = array [image_region] of visir_image;

type visir_set_sequence = array [time] of visir_set;

type histogram = array [temperature] of count;
```

```
type histogram_set =
  array [image_region] of
    structure {
      .hist_location = earth_location;
      .hist_histogram = histogram;
    };
```

Data objects of type *visir_image* are two-dimensional images of *temperature* and *brightness* values, indexed by *earth_location* values. The cloud discrimination algorithm partitions images into regions, and a data object of type *visir_set* is an image with partitions indexed by *image_region* values. A data object of type *visir_set_sequence* is a *time* sequence of partitioned images. A *histogram* data object attaches a frequency *count* to a set of *temperature* values, and a *histogram_set* object contains a *histogram* and an *earth_location* value for each *image_region* value.

Type definitions can be used to attach ancillary values to data objects. For example, the following type defintions provide a way to attach a sensor name, a satellite sub-point, and a table of errors as a function of temperature, to each image in a time sequence:

```
type sensor_name = string;
type rms_error = real;

type visir_sequence =
  array [time] of
    structure {
      .vs_sensor = sensor_name;
      .vs_sub_point = earth_location;
      .vs_error = array [temperature] of rms_error;
      .vs_visir = visir_image;
    };
```

4. DATA OBJECTS

Define $D(t)$ as the set of data objects of a type $t \in T$, sometimes called the "domain" of a data type. The domains of scalar types are determined from the domains of their primitive types, by $D(s)=D(P(s))$. The domain of the primitive type *int* is the union of a set of finite sub-domains, each an interval of integers, as follows:

$$D(\mathrm{int}_{i,j}) = \{k | i \le k \le j\}$$

$$D(\mathrm{int}) = \{missing\} \cup \bigcup_{i \le j} D(\mathrm{int}_{i,j})$$

where i, j and k are integers and the *missing* value indicates the lack of information (the use of special "missing data" codes is common in remote sensing algorithms). The domain of the primitive type *real* is the union of a set of finite sub-domains, each a set of half-open intervals, as follows:

$$D(real_{f,i,j,n}) = \{f(k/2^n), f((k+1)/2^n)) | i \le k \le j\}$$

$$D(real) = \{missing\} \cup \bigcup_{f \in Fld} \bigcup_{i \le j, n \ge 0} D(real_{f,i,j,n})$$

where i, j, k and n are integers and Fld is a set of increasing continuous bijections from **R** (the set of real numbers) to **R**; the functions in Fld provide non-uniform sampling of *real* values. The domains $D(real2d)$, $D(real3d)$ and $D(string)$ are similarly defined as the unions of finite sub-domains.

$D((s \to t))$ is defined as the union of a set of function spaces, rather than as the single space of functions from $D(s)$ to $D(t)$, as follows:

$$D((s \to t)) = \{missing\} \cup \bigcup_{subs} (D(s_{subs}) \to D(t))$$

where *subs* ranges over the finite sub-domains of the scalar domain $D(s)$, and $(D(s_{subs}) \to D(t))$ denotes the set of all functions from the set $D(s_{subs})$ to the set $D(t)$. Every array object in $D((s \to t))$ contains a finite set of values from $D(t)$, indexed by values from one of the finite sub-domains of $D(s)$.

The domains of tuple types are defined by:

$$D((t_1,...,t_n)) = \{missing\} \cup D(t_1) \times ... \times D(t_n)$$

Each domain $D(t)$ has a lattice structure, with the *missing* value as its least element. The half-open intervals in $D(real)$ are approximations to values in **R** and are ordered by the inverse of set inclusion; that is, in the lattice structure, an interval is "less" than its sub-intervals. Values in $D(real2d)$ and $D(real3d)$ form similar lattices and are approximations to values in $\mathbf{R}^2$ and $\mathbf{R}^3$. The lattice structure can be extended to array and tuple types.

The lattice structure of domains, and the definition of array domains as unions of function spaces, provide a formal basis for interpreting array data objects whose indices have primitive types *real*, *real2d* or *real3d* as finite samplings of functions over **R**, $\mathbf{R}^2$ or $\mathbf{R}^3$. For example, a satellite image is a finite sampling of a continuous radiance field. The VIS-AD programming language allows arrays to be indexed by *real*, *real2d* and *real3d* values. Navigation (earth alignment) and calibration (radiance normalization) for satellite images can be implemented by appropriately defined sub-domains of $D(real2d)$ and $D(real)$, so that raw satellite images can be accessed directly in terms of latitude, longitude and temperature.

Physical variables range over infinite sets of values, such as the set **R** of real numbers. However, values must be stored in computers using a finite number of bits, and thus are constrained to range over finite sets of values, such as the set of 32-bit floating point numbers. These are finite samplings of infinite value sets. In most programming languages, the finite samplings for variables

are determined by the type of a variable (for example, *real* or *double* in C). In the VIS-AD programming language, however, the finite samplings are specified as part of the data object. A scalar domain $D(s)$ is a union of a set of finite sub-domains, and each sub-domain is a different finite sampling of the infinite set that is the completion of the lattice $D(s)$ (for example, R is the completion of D(*real*)).

The pixel locations in a satellite image form a finite sampling of an infinite set of points on the earth. It is usual to store image data as arrays. Since arrays indices are identified as scalar types in VIS-AD, and since the finite sampling of the array index is part of an array data object, the navigation information for the satellite image is part of the image data object. Similarly the 256 radiance values of an 8-bit pixel are a finite sampling of temperatures, so the calibration information for a satellite image is also part of the image data object. Thus the domains for VIS-AD data objects provide a uniform way to manage these ancillary information as part of the data objects themselves.

One simple consequence of the VIS-AD data domain structure is support for variable length arrays. That is, the sizes of arrays are not fixed in their declarations, but may vary. Thus arrays can be used to model list structures. For example, a map boundary can be defined with the following data types:

```
type list_index = int;

type map_boundary =
  array [list_index] of earth_location;
```

A *map_boundary* data object is a variable length array of *earth_location* points. Thus, although VIS-AD does not explicitly support linked data structures, it can easily model simple list structures.

The VIS-AD support for *missing* data is motivated by its use for managing remote sensing data. However, *missing* data can also be used as a data structuring tool. For example, a data object *image_area* of type *visir_image* can be used to represent an arbitrarily shaped image region simple by setting

```
image_area[earth_location] = missing;
```

for all values of *earth_location* not in the image region.

Thus navigation, calibration and missing data indicators can be built into the values of data objects, variable length arrays can be used to model list structures, and missing data can be used as a data structuring tool. Combined with the flexibility of type definitions illustrated in Section 4, VIS-AD provides very powerful tools for managing earth science data.

5. ACCESS TO DATA OBJECTS

The VIS-AD programming language provides a transparent way to access the finite sampling information of data objects. For example, if *goes* is an object of type *visir_image* and *loc* is an object of type *earth_location*, then VIS-AD evaluates the expression *goes[loc]* as:

> if *loc* is outside the range of the finite sampling of index values of *goes*, then evaluate *goes[loc] = missing*
> otherwise, resample *loc* to the actual index value *loc'* of the *goes* array closest to *loc*, and evaluate *goes[loc] = goes[loc']*

Thus array accesses may evaluate to *missing*. VIS-AD provides a transparent way to manage operations on *missing* values. If OP is a binary arithmetical operator (+, -, * or /) and *val1* and *val2* are expressions with scalar values, then VIS-AD evaluates the expression *val1* OP *val2* to *missing*:

> if *val1* = *missing* or
> if *val2* = *missing* or
> if OP == / and *val2* == 0

These evaluation rules make it easy to combine data from different sources, with out the need to explicitly remap one set of data to the projection of the other. For example, let *goes_west* and *goes_east* be two data objects of type *visir_image*, from the west and east GOES satellites respectively. The pixels in these images are not co-located, but the difference of these images can be calculated quite simply by:

```
foreach (loc in goes_west) {
  goes_west[loc] = goes_west[loc] - goes_east[loc];
};
```

where *loc* is a data object of type *earth_location*. Inside the *foreach* loop, the value of *loc* varies over all the array index values of the array *goes_west*. The values of *goes_east* are resampled to the index locations of *goes_west*, and the difference of these image arrays evaluates to *missing* wherever they do not overlap.

VIS-AD also provides a simple means to access sub-objects of data objects. For example, if *hset* is a data object of type *histogram_set* and if *reg* is a data object of type *image_region*, then the expression *hset[reg].hist_histogram* evaluates to a data object of type *histogram*.

VIS-AD includes functions for converting objects between their internal storage formats and external formats suitable for storage in disk files and for transmission to other processes or across computer networks. Both the internal and external object formats use memory efficiently. Numerical values are stored as scaled integers rather than as floating point numbers, and use 8-bit or 16-bit integers wherever possible. These formats minimize use of disk storage and communications bandwidth. The absence of any floating point values eliminates the need for converting

data objects between machine architectures (except for possible problems with big-endian versus small-endian machines, and machines that use non-ascii text).

6. HIGH-LEVEL FUNCTIONS

VIS-AD supports calls to three kinds of functions. Internal functions are implemented in the VIS-AD programming language, and users writing VIS-AD programs are free to define as many internal functions as they need. Intrinsic functions are implemented as part of the VIS-AD system and should be viewed by users as part of the language (like the MAX function in FORTRAN). External functions are implemented in C or FORTRAN, and give users a way to link their existing programs to VIS-AD.

The VIS-AD system includes a variety of intrinsic functions for transferring McIDAS data structures (for example, image and grid files) into VIS-AD data objects, and for analyzing data. We are constantly adding new intrinsic functions to the system. Analysis functions are currently defined for:

- Remapping two- and three-dimensional images and grids.
- Low-pass filtering one-, two- and three-dimensional data.
- Calculating histograms of data arrays.
- Finding clusters in histograms.
- Finding percentiles of histograms.
- Selecting regions of arrays with values in selected ranges.
- Boolean operations on regions of arrays.

VIS-AD external functions, written by the user in C or FORTRAN, provide a way for users to access data sets stored in any format, and a way for users to link to their existing analysis functions.

7. CONCLUSION

The purpose of this paper is to point out that VIS-AD contains many powerful functions for managing earth science data. These include:

- An easy way for users to define new data structures, such as images, multi-spectral images, image sequences, histograms, spatial regions, region boundaries, etc. These flexible data types also let users define a variety of ancillary data as part of their data types.
- An easy way to access data objects and their sub-components a programming language, and an easy way to write functions for analyzing those data.
- Uniform mechanisms for management of all data structures.
- A special *missing* data indicator that can be used for the value of any data object or sub-object.
- A uniform mechanism for managing finite samplings of continuous quantities, as for example, the way that satellite navigation and calibration finitely sample earth locations and temperatures.
- A easy way for users to write C or FORTRAN programs for converting data between VIS-AD data objects and other systems.
- A mechanism for storing data objects in disk files or for transmitting data objects across networks.

Thus, the most difficult functions for managing complex earth science data already exist in VIS-AD. In order for VIS-AD to function as a true data management system, it needs to include functions for:

- Immediate mode commands. Currently, all functions are called by a running VIS-AD program. Users need a way to invoke functions one at a time by typing commands.
- Managing data objects in disk files. There should be commands for transfering data objects between disk files and memory, for listing objects in disk files, and possibly for retrieving sub-objects of objects stored in disk files (since data objects may be large).

After these functions are implemented, VIS-AD will be a powerful earth science data manager, in addition to its original role for visualizing the behavior of scientific algorithms.

8. REFERENCES

Hibbard, W., C. Dyer and B. Paul, 1992a; A development environment for data analysis algorithms. Preprints, Conf. Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology. Atlanta, American Meteorology Society. 101-107.

Hibbard, W., C. Dyer, and B. Paul, 1992b; Display of scientific data structures for algorithm visualization. Accepted for Visualization '92, IEEE.